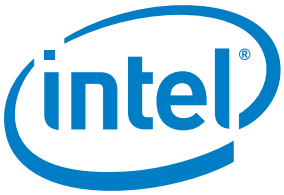

Analyzing potential bounds check bypass vulnerabilities

White Paper

Revision 002
July 2018

Document Number: 337879-002

Contents

Revision History

Date	Revision	Description
July 10, 2018	001	Initial release.
July 26, 2018	002	Added mitigation information for RSB exploits.

§

1.0 Introduction

Side channel methods are techniques that may allow a malicious actor to gain information through observing the system, such as measuring microarchitectural properties about the system. For an introduction to speculation and these methods, see the Intel Analysis of Speculative Execution Side Channels white paper (https://software.intel.com/sites/default/files/managed/b9/f9/336983-Intel-Analysis-of-Speculative-Execution-Side-Channels-White-Paper.pdf), Speculative Execution Side Channel Mitigations white paper (https://software.intel.com/sites/default/files/managed/c5/63/336996-Speculative-Execution-Side-Channel-Mitigations.pdf), and refer to the security research findings page on intel.com (https://www.intel.com/content/www/us/en/architecture-and-technology/facts-about-side-channel-analysis-and-intel-products.html).

Bounds check bypass represents a broad class of vulnerabilities. This document examines common instances of these vulnerabilities, including the *bounds check bypass store* variant, but should not be considered a comprehensive list.

This document describes how to analyze potential *bounds check bypass* and *bounds check bypass store* vulnerabilities found by static analysis tools or manual code inspection and presents mitigation techniques that may be used. This document does not include any actual code from any real product or open source release, nor does it discuss or recommend any specific analysis tools.

Bounds check bypass takes advantage of the speculative execution used in processors to achieve high performance. To avoid the processor having to wait for data to arrive from memory, or for previous operations to finish, the processor may speculate as to what will be executed. If it is incorrect, the processor will discard the wrong values and then go back and redo the computation with the correct values. At the program level this speculation is invisible, but because instructions were speculatively executed they might leave hints that a malicious actor can measure, such as which memory locations have been brought into cache.

Using the bounds check bypass method, malicious actors can use code gadgets ("confused deputy" code) to infer data values that have been used in speculative operations. This presents a method to access data in the system cache and/or memory that the malicious actor should not otherwise be able to read. The bounds check bypass store variant makes an additional range of vulnerabilities possible by targeting variables on the stack, function pointers, or return addresses. This allows malicious actors to influence variables used later in speculative execution or to direct speculative execution to other areas of code, where malicious actors could then observe system behavior.

These two methods of attack can be mitigated by modifying software through the insertion of a serializing instruction to constrain speculation in confused deputies. Such instructions ensure that all instructions in the processor's instruction pipeline up to the speculation barrier resolve before any later instructions in the pipeline can execute. This prevents the processor from speculatively accessing data that the user should not have access to, because no speculative operations can run until the bounds check operation completes.

The focus of this document is analyzing code written in C and C++, in particular code running at a higher privilege level than the malicious actor's code. Similar techniques apply to assembly language and many other compiled languages. JITs and languages where runtime safety is imposed by the compiler are discussed in more depth in the Managed Runtime Speculative Execution Side Channel Mitigations (https://software.intel.com/sites/default/files/managed/7c/4a/Managed-Runtime-Speculative-Execution-Side-Channel-Mitigations.pdf) white paper.

2.0 Where mitigations are relevant

Bounds check bypass mitigations are not generally relevant if your code doesn't have secrets that the user shouldn't be able to access. For example, a simple image viewer probably contains no meaningful secrets that should be inaccessible to software it interacts with. The user of the software could potentially use bounds check bypass attacks to access the image, but they could also just hit the save button.

On the other hand, an image viewer with support for secure, encrypted content with access authorized from a central system might need to care about bounds check bypass because a user may not be allowed to save the document in normal ways. While the user can't save such an image they can trivially photograph the image and send the photo to someone, so protecting the image may be less important. However, any keys are likely to be far more sensitive.

There are also clear cases like operating system kernels, firmware (refer to the *Host Firmware Speculative Execution Side Channel Mitigation* white paper) and managed runtimes (for example, Javascript* in web browsers) where there is both a significant interaction surface between differently trusted code, and there are secrets to protect.

Whether to apply mitigations, and what areas to target has to be part of your general security analysis and risk modelling, along with conventional security techniques, and resistance if appropriate to timing and other non-speculative side channel attacks. Bounds check bypass mitigations have performance impacts, so they should only be used where appropriate.

3.0 Identifying bounds check bypass vulnerabilities

3.1 Common attributes for bounds check bypass vulnerabilities

Bounds check bypass code sequences have some common features: they generally operate on data that is controlled or influenced by a malicious actor, and they all have some kind of side-effect that can be *observed* by the malicious actor. In addition, the processor's speculative execution sequence executes in a way which would be thrown away in a normally retired execution sequence. In bounds check bypass store variants, data is speculatively written at locations that would be out of bounds under normal execution. That data is later speculatively used to execute code and cause observable side-effects, creating a side channel.

3.2 Loads and stores

A vulnerable code fragment forming a disclosure gadget is made up of two elements. The first is an array or pointer dereference that depends upon an untrusted value, for example, a value from a potentially malicious application. The second element is usually a load or store to an address that is dependent upon the value loaded by the first element. Refer to Microsoft's* blog (https://blogs.technet.microsoft.com/srd/2018/03/15/mitigating-speculative-execution-side-channel-hardware-vulnerabilities/) for further details.

As bounds check bypass is based upon speculation, code can be vulnerable even if that untrusted value is correctly tested for bounds before use.

The classic general example of such a sequence in C is:

```
if (user_value >= 0 && user_value < LIMIT) {
        x = table[user_value];
        node = entry[x];
} else
        return ERROR;
```

For such a code sequence to be vulnerable, both elements must be present. Furthermore the untrusted value must be under the malicious actor's control.

When the code executes, the processor has to decide if the `user_value < LIMIT` conditional is true or false. It remembers the processor register state at this point and speculates (makes a guess) that `user_value` is below `LIMIT` and begins executing

instructions as if this were true. Once the processor realizes it guessed incorrectly, it throws away the computation and returns an error. The attack relies upon the fact that before it realizes the guess was incorrect, the processor has read both `table[user_value]`, pointing into memory beyond the intended limit, and has read `entry[x]`. When the processor reads `entry[x]`, it may bring in the corresponding cache line from memory into the L1 cache. Later, the malicious actor can time accesses to this address to determine whether the corresponding cache line is in the L1 data cache. The malicious actor can use this timing to discover the value `x`, which was loaded from a location specified by a malicious actor.

The two components that make up this vulnerable code sequence can be stretched out over a considerable distance and through multiple layers of function calls. The processor can speculatively execute many instructions—a number sufficient to pass between functions, compilation units, or even software exception handlers such as `longjmp` or `throw`. The processor may speculate through locked operations, and use of `volatile` will not change the vulnerability of the code being exploited.

There are several other sequences that may leak information. Anything that tests some property of a value and loads or stores according to the result may leak information. Depending upon the location of `foo` and `bar`, the example below might be able to leak bit 0 of arbitrary data.

```
if (user_value >= LIMIT)
      return ERROR;
   x = table[user_value];
   if (x & 1)
      foo++;
   else
      bar++;
```

When evaluating code sequences for vulnerability to bounds check bypass, the critical question is whether different behavior could be observed as a property of `x`.

This question can be very challenging to answer from code inspection, especially when looking for any specific code pattern. For instance, if a value is passed to a function call, then that function call must be inspected to ensure it does not create any observable interactions. Consider the following example:

```
if (user_value >= LIMIT)
      return ERROR;
   x = lengths[user_value];
   if (x)
```

```
memset(buffer, 0, 64 * x);
```

Here, x influences how much memory is cleared by `memset()` and might allow the malicious actor to discern something about the value of x from which cache lines the speculatively executed memset touches.

Remember that conditional execution is not just `if`, but may also include `for` and `while` as well as the C ternary (`?:`) operator and situations where one of the values is used to index an array of function pointers.

3.3 Typecasting and indirect calls

Typecasting can be a problematic area to analyze and often conceals real examples that can be exploited. This is especially challenging in C++ because you are more likely to have function pointers embedded in objects and overloaded operators that might behave in type-dependent fashion.

Two classes of typecasting problems are relevant to bounds check bypass attacks:

1. *Code/data mismatches*. Speculation causes "class `Foo`" code to be speculatively executed on "class Bar" data using gadgets supplied with `Foo` to leak information about `Bar`.
2. *The type confusion is combined with some observable effect, like the load/store effects discussed above*. For example, if `Foo` and `Bar` are different sizes, a malicious actor might be able to learn something about memory past the end of `objects[]` using something like the example below.

```
type = objects[index];

if (index >= len)

   return -EINVAL;

if (type == TYPE_FOO)

   memset(ptr, 0, sizeof(Foo));

else

   memset(ptr, 0, sizeof(Bar));
```

Take care when considering any code where a typecast occurs based upon a speculated value. The processor might guess the type incorrectly and speculatively execute instructions based on that incorrect type. Newer processors that enable Intel® OS Guard, also known as Supervisor-Mode Execution Prevention (SMEP), will prevent ring 0 code from speculatively executing ring 3 code. All major operating systems (OSes) enable SMEP support by default if the hardware supports it. Older processors however, might speculate the type incorrectly, load data that the processor thinks are

function pointers, or speculate into lower addresses that might be directly controlled by a malicious actor.

For example:

```
if (flag & 4)
      (Foo *)ptr->process(x);
else
      (Bar *)ptr->process(x);
```

If the `Foo` and `Bar` objects are different and have different memory layouts, then the processor will speculatively fetch a pointer offset of `ptr` and branch to it.

Consider the following example:

```
int call; /* from user */
if (call >= 0 && call < MAX_FUNCTION)
      function_table[call](a,b,c);
```

On first analysis this code might seem safe. We reference `function_table[call]`, but `call` is the user's own, known value. However, during speculative execution, the processor might incorrectly speculate through the `if` statement and speculatively execute invalid addresses. Some of these addresses might be mapped to user pages in memory, or might contain values that match suitable gadgets for ROP attacks.

A less obvious variant of this case is switch statements. Many compilers will convert some classes of switch statement into jump tables. Refer to the following example code:

```
switch(x) {
case 0: return y;
case 1: return z;
...
default: return -1;
}
```

Code similar to this will often be implemented by the compiler as shown:

```
if (x < 0 || x > 2) return -1;
goto  case[x];
```

Therefore when using `switch()` with an untrusted input, it might be appropriate to place an `lfence` before the switch so that x has been fully resolved before the implicit bounds check.

3.4 Speculative loops

A final case to consider is loops that speculatively overrun. Consider the following example:

```
while (++x < limit) {
   y = u[x];
   thing(y);
}
```

The processor will speculate the loop condition, and often speculatively execute the next iteration of the loop. This is usually fine, but if the loop contains code that reveals the contents of data, then you might need to apply mitigations to avoid exposing data beyond the intended location of the loop. This means that even if the loop limit is properly protected before the processor enters the loop, unless the loop itself is protected, the loop might leak a small amount of data beyond the intended buffer on the speculative path.

3.5 Disclosure gadgets

In addition to the load and store disclosure gadget referenced above, there may be additional gadgets based on the microarchitectural state. For example, using certain functional blocks, such as Intel® Advanced Vector Extensions (Intel® AVX), during speculative execution may affect the time it takes to subsequently use the block due to factors like the time required to power-up the block. Malicious actors can use a disclosure primitive to measure the time it takes to use the block. An example of such a gadget is shown below:

```
if (x > sizeof(table))
      return ERROR;
   If (a[x].op == OP_VECTOR)
      avx_operation(a[x]);
   else
      integer_operation(a[x]);
```

4.0 Bounds check bypass store attacks

The *bounds check bypass store* method makes an additional range of vulnerabilities possible. By targeting the stack or function pointers, it is possible to manipulate the speculative code flow of the processor in a way that directs execution to speculative code which the processor may not expect to be called with untrusted user data. This provides a way for a malicious actor to point speculative execution in more specific areas where they can measure operations in order to collect secrets.

In the following example, a speculative store allows the malicious actor to speculatively overwrite any variables, temporary values, or function pointers that will be called by the processor with data under the malicious actor's control. The malicious actor can also speculatively modify return addresses on the stack to make the processor speculatively execute disclosure code present in the system. As the instruction and data cache are separated, this attack cannot directly target code, only things like function pointers or return addresses.

```
int function(int user_index, unsigned long user_key) {
      unsigned long data[8];

if (user_index < 8)
         data[user_index] = user_key;
      else
         return -1;

sort_table(data);

      return 0;
}
```

The example above does not by itself allow a bounds check bypass attack. However, it does allow the attack to speculatively modify memory, and therefore could potentially be used to chain attacks. For example a speculative write to the return address could cause the final `return 0` to speculatively return to a user-controlled disclosure gadget.

Where the compiler has spilled variables to the stack, the store can also be used to target those spilled values and speculatively modify them to enable another attack to follow. An example of this would be by targeting the base address of an array

dereference or the limit value. For further information, refer to the *Variant 1: Bounds Check Bypass* section of the Intel Analysis of Speculative Execution Side Channels white paper (https://software.intel.com/sites/default/files/managed/b9/f9/336983-Intel-Analysis-of-Speculative-Execution-Side-Channels-White-Paper.pdf).

A second variant of this method can occur where a user value is being copied into an array, either on the stack or adjacent to function pointers. As discussed previously, the processor may speculatively execute a loop more times than is actually needed. If this loop moves through memory writing malicious actor-controlled values, then the malicious actor may be able to speculatively perform a buffer overrun attack.

```
int filltable(uint16_t *from)
{
      uint16_t buffer[64];
      int i;

      for (i = 0; i < 64; i++)
            buffer[i] = *from++;
}
```

In some cases, the example above might speculatively copy more bytes than 64 into the array, changing the return address speculatively used by the processor so that it instead returns to a user controlled gadget.

As the execution is speculative, some processors will allow speculative writes to read-only memory, and will reuse that data speculatively. Therefore, while placing function pointers into write-protected space is a good general security mitigation, doing so is not sufficient mitigation in this case.

5.0 Software mitigations for bounds check bypass and bounds check bypass store

Software can insert a speculation stopping barrier between a bounds check and a later operation that could cause a speculative side channel. The LFENCE instruction, or any serializing instruction, can serve as such a barrier. These instructions suffice regardless of whether the bounds checking is implemented using conditional branches or through the use of boundchecking instructions (BNDCL and BNDCU) that are part of the Intel® Memory Protection Extensions (Intel® MPX).

The LFENCE instruction and other serializing instructions (refer to Chapter 8.3, Volume 3a of the Intel Software Developers Manual (https://software.intel.com/en-us/articles/intel-sdm)) ensure that no later instruction will execute, even speculatively, until all prior instructions have completed locally. This prevents the processor from speculatively accessing data that might be out-of-bounds for the user, because no speculative operations can run until this bounds check completes. This essentially creates a barrier where speculative execution cannot take place in locations not allowed for that user. The LFENCE instruction has lower latency than the serializing instruction execution and thus is recommended.

5.1 LFENCE

The main mitigation for bounds check bypass is through use of the LFENCE instruction. The LFENCE instruction does not execute until all prior instructions have completed locally, and no later instruction begins execution until LFENCE completes. Most vulnerabilities identified in the Identifying vulnerabilities section can be protected by inserting an LFENCE instruction; for example:

```
if (user_value >= LIMIT)
   return ERROR;
lfence();
x = table[user_value];
node = entry[x]
```

Where `lfence()` is a compiler intrinsic or assembler inline that issues an LFENCE instruction and also tells the compiler that memory references may not be moved across that boundary. The LFENCE ensures that the loads do not occur until the condition has actually been checked. The memory barrier prevents the compiler from reordering references around the LFENCE, and thus breaking the protection.

5.1.1 Placement of LFENCE

To protect against speculative timing attacks, place the LFENCE instruction after the range check and branch, but before any code that consumes the checked value, and before the data can be used in a gadget that might allow measurement.

For example:

```
if (x > sizeof(table))
      return ERROR;
   lfence();
   If (a[x].op == OP_VECTOR)
      avx_operation(a[x]);
   else
      integer_operation(a[x]);
```

Unless there are specific reasons otherwise, and the code has been carefully analyzed, Intel recommends that the LFENCE is always placed after the range check and before the range checked value is consumed by other code, particularly if the code involves conditional branches.

5.2 Bounds clipping

Other instructions such as CMOVcc, AND, ADC, SBB and SETcc can also be used to help prevent bounds check bypass by constraining speculative execution on current family 6 processors (Intel® Core™, Intel® Atom™, Intel® Xeon® and Intel® Xeon Phi™ processors). However, these instructions may not be guaranteed to do so on future Intel processors. Intel intends to release further guidance on the usage of instructions to constrain speculation in the future before processors with different behavior are released.

Memory disambiguation (described in the Speculative Execution Side Channel Mitigations white paper (https://software.intel.com/sites/default/files/managed/c5/63/336996-Speculative-Execution-Side-Channel-Mitigations.pdf) can theoretically impact such speculation constraining sequences when they involve a load from memory.

This approach can avoid stalling the pipeline as LFENCE does.

At the simplest:

```
unsigned int user_value;
```

```
if (user_value > 255)
   return ERROR;
x = table[user_value];
```

Can be made safe by instead using the following logic:

```
volatile unsigned int user_value;


if (user_value > 255)
   return ERROR;
x = table[user_value & 255];
```

This works for powers of two array lengths or bounds only. In the example above the table array length is 256 (2^8), and the valid index should be <= 255. Take care that the compiler used does not optimize away the & 255 operation. For other ranges, it's possible to use CMOVcc, ADC, SBB, SETcc, and similar instructions to do verification.

Although this mitigation approach can be faster than other approaches it is not guaranteed for the future. Developers who cannot control which CPUs their software will run on (such as general application, library, and SDK developers) should not use this mitigation technique. Intel intends to release further guidance on how to use serializing instructions to constrain speculation before future processors with different behavior are released.

Both of these techniques can be applied to function call tables, while the LFENCE approach is generally the only technique that can be used when typecasting.

5.3 Multiple branches

When using mitigations, particularly the bounds clipping mitigations, it is important to remember that the processor will speculate through multiple branches. Thus, the following code is not safe:

```
int *key;
int valid = 0;


if (input < NUM_ENTRIES) {
      lfence();
      key = &table[input];
      valid = 1;
```

```
}
….
if (valid)
      *key = data;
```

In this example, although the mitigation is applied correctly when the processor speculates that the first condition is valid, no protection is applied if the processor takes the out-of-range value and then speculates that `valid` is true on the other path. In this case it will probably expose the contents of a random register, although not in an easy to measure fashion.

Preinitializing `key` to `NULL` or another safe address will also not reliably work, as the compiler can eliminate the `NULL` assignment because it can never be used non-speculatively. In such cases it may be more appropriate to merge the two conditional code sections and put the code between them into a separate function that is called on both paths. Or you could add `volatile` to `key` and assign it to `NULL`—forcing the assignment to occur with `volatile`, or to add `lfence` before the final assignment.

5.4 Existing compiler mitigations

Existing compiler protections against buffer overwrites of return addresses, such as stack canaries, provide some resistance to speculative buffer overruns. In situations where a loop speculatively overwrites the return address it will also speculatively trigger the stack protection diverting the speculative flow. Stack canaries alone are not sufficient to protect from bounds check bypass attacks.

5.5 Additional compiler mitigations

5.5.1 Microsoft* Visual Studio* 2017 mitigations

The Microsoft Visual Studio* 2017 Visual C++ compiler toolchain includes support for the `/Qspectre` flag, which may automatically add mitigation for some bounds check bypass vulnerabilities. For more information and usage guidelines, refer to Microsoft's public blog (https://blogs.msdn.microsoft.com/vcblog/2018/01/15/spectre-mitigations-in-msvc/) and the Visual C++ /Qspectre option page (https://docs.microsoft.com/en-us/cpp/build/reference/qspectre) for further details.

5.5.2 LFENCE in Intel® Fortran Compiler

You can insert an LFENCE instruction in Fortran applications as shown in the example below. Implement the following subroutine, which calls `_mm_lfence()` intrinsics:

```
interface
        subroutine for_lfence() bind (C, name = "_mm_lfence")
            !DIR$ attributes known_intrinsic, default ::
for_lfence
        end subroutine for_lfence
    end interface


    if (untrusted_index_from_user .le. iarr1%length) then
        call for_lfence()
        ival = iarr1%data(untrusted_index_from_user)
        index2 = (IAND(ival,1)*z'100') + z'200'
        if(index2 .le. iarr2%length)
            ival2 = iarr2%data(index2)
    endif
```

The LFENCE intrinsic is supported in the following Intel compilers:

- Intel C++ Compiler 8.0 and later for Windows*, Linux*, and macOS*.
- Intel Fortran Compiler 14.0 and later for Windows, Linux, and macOS.

5.5.3 Compiler-driven automatic mitigations

Across the industry, there is interest in mitigations for bounds check bypass vulnerabilities that are provided automatically by compilers. Developers are continuing to evaluate the efficacy, reliability, and robustness of these mitigations and to determine whether they are best used in combination with, or in lieu of, the more explicit mitigations discussed above.

6.0 Operating system mitigations

Where possible, dedicated operating system programming APIs should be used to mitigate bounds check bypass instead of using open-coded mitigations. Using the OS-provided APIs will help ensure that code can take advantage of new mitigation techniques or optimizations as they become available.

6.1 Linux* kernel

The current Linux* kernel mitigation approach to bounds check bypass is described in the *speculation.txt* (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/speculation.txt) file in the Linux kernel documentation. This file is subject to change as developers and multiple processor vendors determine their preferred approaches.

`ifence()`: on x86 architecture, this issues an LFENCE and provides the compiler with the needed memory barriers to perform the mitigation. It can be used as `lfence()`, as in the examples above. On non-Intel processors, `ifence()` either generates the correct barrier code for that processor, or does nothing if the processor does not speculate.

`array_ptr(array, index, max)`: this is an inline that, irrespective of the processor, provides a method to safely dereference an array element. Additionally, it returns `NULL` if the lookup is invalid. This allows you to take the many cases where you range check and then check that an entry is present, and fold those cases into a single conditional test.

Thus we can turn:

```
if (handle < 32) {
   x = handle_table[handle];
   if (x) {
      function(x);
      return 0;
   }
}
return –EINVAL;
```

Into:

```
x = array_ptr(handle_table, handle, 32);
```

```
if (x == NULL)
   return –EINVAL;
function(*x);
return 0;
```

6.2 Microsoft Windows*

Windows C/C++ developers have a variety of options to assist in mitigating bounds check bypass. The best option will depend on the compiler/code generation toolchains you are using. Mitigation options include manual and compiler assisted.

In mixed-mode compiler environments, where object files for the same project are built with different toolchains, there are varying degrees of mitigation options available. Developers need to be aware of and apply the appropriate mitigations depending on their code composition and appropriate toolchain support dependencies.

As described in section 5.0 Software mitigations for bounds check bypass and bounds check bypass store, we recommend inserting LFENCE instructions (either manually or with compiler assistance) for mitigating bounds check bypass on Windows. The following sections provide details on how to insert the LFENCE instruction using currently available compiler tool chain mechanisms. These mechanisms are (from lowest level to highest level):

- Inline/external assembly
- `_mm_lfence()` compiler intrinsic
- Compiler automatic LFENCE insertion

6.2.1 Inline/external assembly

The Intel® C Compiler and Intel® C++ Compiler provide inline assembly support for 32- and 64-bit targets, whereas Microsoft Visual* C++ only provides inline assembly support for 32-bit targets. Microsoft Macro Assembler* (MASM) or other external, third party assemblers may also be used to insert LFENCE in assembly code.

6.2.2 _mm_lfence() compiler intrinsic

The Intel C Compiler, the Intel C++ Compiler, and the Microsoft Visual C++ compiler all support generating LFENCE instructions for 32- and 64-bit targets using the `_mm_lfence()` intrinsic.

The easiest way for Windows developers to gain access to the intrinsic is by including the *intrin.h* header file that is provided by the compilers. Some Windows SDK/WDK headers (for example, *winnt.h* and *wdm.h*) define the `_mm_lfence( )` intrinsic to avoid inclusion of the compiler *intrin.h*. It is possible that you already have code that locally defines `_mm_lfence( )` as well, or uses an already existing definition for the intrinsic.

6.2.3 LFENCE in C/C++

You can insert LFENCE instructions in a C/C++ program as shown in the example below:

```
#include <intrin.h>
#pragma intrinsic(_mm_lfence)


    if (user_value >= LIMIT)
    {
        return STATUS_INSUFFICIENT_RESOURCES;
    }
    else
    {
        _mm_lfence();   /* manually inserted by developer */
        x = table[user_value];
        node = entry[x];
    }
```

7.0 References

- https://docs.microsoft.com/en-us/cpp/build/reference/qspectre
- https://docs.microsoft.com/en-us/visualstudio/releasenotes/vs2017-relnotes#top-issues-fixed-in-156
- https://docs.microsoft.com/en-us/visualstudio/releasenotes/vs2017-relnotes-v15.5#15.5.5
- https://docs.microsoft.com/en-us/visualstudio/releasenotes/vs2017-relnotes-v15.0#15.0.26228.23
- https://support.microsoft.com/en-us/help/4073757/protect-your-windows-devices-against-spectre-meltdown
- https://portal.msrc.microsoft.com/en-US/security-guidance/advisory/ADV180002
- https://cloudblogs.microsoft.com/microsoftsecure/2018/01/09/understanding-the-performance-impact-of-spectre-and-meltdown-mitigations-on-windows-systems/
- https://blogs.msdn.microsoft.com/vcblog/2018/01/15/spectre-mitigations-in-msvc/
- https://software.intel.com/sites/default/files/managed/c5/63/336996-Speculative-Execution-Side-Channel-Mitigations.pdf
- https://software.intel.com/sites/default/files/managed/b9/f9/336983-Intel-Analysis-of-Speculative-Execution-Side-Channels-White-Paper.pdf
- https://www.intel.com/content/www/us/en/architecture-and-technology/facts-about-side-channel-analysis-and-intel-products.html
- https://software.intel.com/en-us/articles/intel-sdm